Supercharging Plant Configurations using Z3

Nikolaj Bjørner, Max Levatich, Nuno P. Lopes, Andrey Rybalchenko,
Chandrasekar Vuppalapati

Microsoft

Abstract. We describe our experiences using Z3 for synthesizing and optimizing next generation plant configurations for a car manufacturing company[1]. Our approach leverages unique capabilities of Z3: a combination of specialized solvers for finite domain bit-vectors and uninterpreted functions, and a programmable extension that we call constraints as code. To optimize plant configurations using Z3, we identify useful formalisms from Satisfiability Modulo Theories solvers and integrate solving capabilities for the resulting non-trivial optimization problems.

1. Introduction

The *digital transformation* is widely recognized as an ongoing seismic shift in today's industries. It encompasses an integration of software technologies in every aspect of a business. AI advances, overwhelmingly dominated by deep machine learning, are widely hailed as pivotal to this shift. Meanwhile, advances in symbolic reasoning, exemplified by Microsoft's Z3 symbolic solver for automated reasoning, have powered automated programming and analysis engines in the past decade. They have been transforming software engineering life cycles by enabling tools for ensuring strong provable guarantees and automatically synthesizing code and configurations. Likewise, digital transformations in the car industry are powering driving experiences. With new models and factories being churned out at a brisk pace, there is an urgent need for automating and optimizing production plants to increase the pace of production while reducing costs and resource requirements. The organization of production assembly lines involves a combination of hundreds of assembly stations and thousands of operators completing tens of thousands tasks with tens of thousands different tools available. Some tasks must be completed in sequential order, some stations may not be able to service tasks with conflicting requirements, only a subset of available operators may be able to work on a given task, and all tasks are packed into stringent timing bounds on each station.

Planning large scale production lines is thus becoming a complex *inhuman* puzzle, that at best takes weeks for an extensively trained expert to solve manually. A manual assignment of tasks comes with no automatic assurance of optimality and with no easy way to explore alternatives. Our experiences with Z3

[1] The views expressed in this writing are our own. They make no representation on behalf of others.

are primarily based on software analysis, verification and synthesis. Z3 is a default tool when it comes to applications around translation validation, symbolic execution, and program verification. It has taken inhuman tasks out of network verification in Azure's operations [21], invoking $O(1B)$ (small) Z3 queries a day, verifying compiler optimizations [23, 24], and finding or preventing security vulnerabilities in complex systems code [6, 17].

Are techniques that have been tested in the area of software analysis usable for production line scheduling? We have some partial answers that indicate the underlying technologies in Z3 can be put to good use in combinatorial domains. The Dynamics product configurator tool [25] ships using Z3 for solving product configuration tasks. It uses Z3 to enumerate consequences: when an operator fixes a fabric of a sofa, the color choices narrow and Z3 integrates a custom optimized consequence finding module built tightly with it's Conflict Driven Clause Learning, CDCL [34], engine. The production plant design scenario is very different from the Dynamics use case, though. The model is complex and does not fit within a commoditized environment. It involves solving multi-knapsack problems with complex side-constraints. We describe our experiences using Z3 for automating next generation production plant designs of a car manufacturing company. Our journey so far involves a combination of *deep cleaning*, thus the activities involved with formalizing a complex model and in the process identifying and fixing data-entry bugs; and *deep solving*, that is, the combination of solver capabilities used to optimize virtual plant configurations. Within the scope of our experiences we pose and test a hypothesis that solving constraints using uninterpreted functions, a base theory of SMT solvers, together with solving for bit-vectors (that capture finite domains), presents a compelling target for multi-knapsack problems with complex side constraints. We argue that uninterpreted functions can be used effectively to encode assignment constraints. Furthermore, solving for these constraints using decision procedures for uninterpreted functions may have a substantial advantage solving MIP or SAT based formulations. To handle multi-knapsack constraints we were compelled to extend Z3 with an interface for user theories: encoding these constraints as *code* appeared readily more viable than supporting custom global constraints.

1.1. Complexity without Perplexity

The complexity and difficulty of the problem we are tackling can be characterized along two dimensions: the complexity inherent in capturing production line models and complexity based on the size of production lines that are solved for. In our case, the production line model requires a few dozen different types of data points. Each type is represented as a database table, and each data-point requires at least three and sometimes more than twenty attributes, where each attribute is represented as a database column. The second dimension of complexity can be measured by the size of database instances that are required to capture production line models. In our case, the order of magnitude along main tables are as follows:

– Stations: $O(100)$. A production line is a sequence of stations. Each station is a collection of around 10 operator positions, of which a subset can be used.
– Operator positions: $O(1K)$.
– Processes: $O(1K)$, each process is a collection of tasks.
– Tasks: $O(10K)$, assigned among the processes.
– Tools: $O(1K)$ are assigned to tasks.

The real killer for straight-forward approaches, though, is that production line constraints involve joining several large tables.

We approach the first dimension through the lens of software engineering methodologies: we are creating a formal model of a production plant and synthesizing optimized configurations. We address the second dimension by describing the technological features we found useful for (efficiently) solving the production line automation.

1.2. Domain Engineering - *Deep Cleaning*

Our approach to production line modeling is very much influenced by concepts and methodologies honed and developed in the software engineering and most specifically formal methods communities. Thus, a starting point is to describe using logical notation a set of domains, functions over the domains, and constraints over the signature. At this stage we seek to delay lower level decisions on how constraints are encoded into a solver. Domain engineering produces a mathematically unambiguous and machine checkable account for production line modeling. We also claim our case is distinguished by some level of complexity: it integrates a combination of many rules and constraints that apply only for special cases. Our experiences with domain engineering falls into two categories. *Domain invariants* are global properties of well-formed production line models. A production line model that violates domain invariants does not correspond to a physical production line. *Domain constraints* capture the solution space of virtual plants. They can take advantage of domain invariants by assuming they have been checked and they don't have to be re-enforced in the constraint encoding.

To summarize, we distinguish between the two categories:

1.2.1. Domain Invariants They describe well-formedness conditions of virtual plant configurations, such as:

– Dependencies between processes are acyclic.
– Stations are connected in a rooted tree comprising of sub-lines.
– Sub-lines may be labeled to force processes within bounds, the same label (called monuments) cannot be used on different branches.

We found that graph visualization tools, in particular MSAGL [26], provided a highly effective way to both communicate assumptions about domain invariants and to uncover violations.

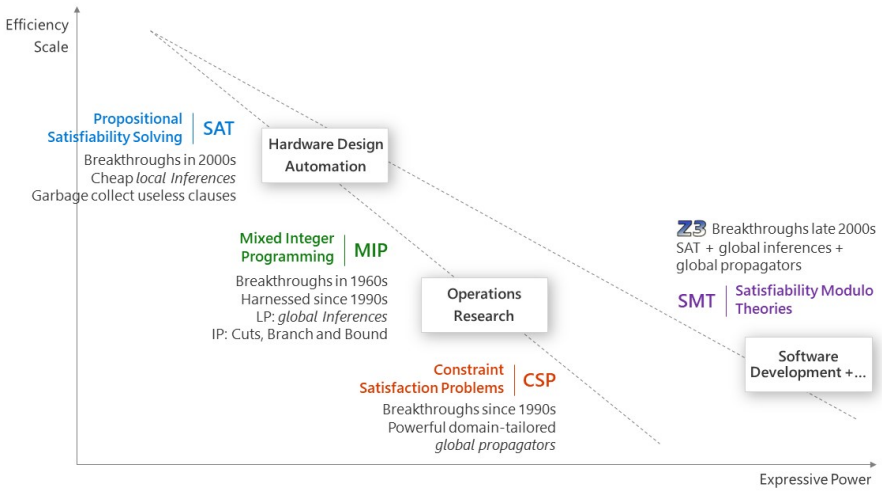

Figure 1. SAT, MIP, CP and SMT

1.2.2. Domain Constraints They encode constraints on valid solutions, and should be:

- Sufficient to capture the solution space of virtual plants.
  - Every solution to hard constraints is a virtual plant solution.
  - Every virtual plant solution has a solution to hard constraints.
- Usable to map constraint violations back into a root cause analysis for data-entry errors.

1.3. Solver Engineering - *Deep Solving*

As the main tool used so far for solving for production lines is Z3, we are going to mainly describe the approach taken relative to Z3 and SMT technologies. Figure 1 suggests a classification of main techniques pursued in the CPAIOR community.

The way to understand the expressiveness/efficiency trad-off is that expressiveness comes with the benefit of handling increasingly succinct ways of capturing constraints, their propositional encoded counter-parts being impractical for SAT solvers. The lower efficiency means that the more expressive solvers use relatively more overhead per succinct constraint than a SAT solver per clause. Each class of domain is labeled by distinguishing features of their mainstream state-of-art solvers. Modern SAT solvers are mainly based on a CDCL architecture that alternates a search for a solution to propositional variables with resolution inferences when a search dead-ends. These inferences rely on a limited set of

premises. Garbage collection of unused derived clauses is a central ingredient to make CDCL scalable. MIP solvers use interior point methods, primal and dual simplex algorithms. Simplex pivoting performs a Gauss-Jordan elimination, which amounts to globally solving for a selected variable with respect to *all* constraints. CP solvers have perfected the art of efficient propagators for global constraints. Effective propagators narrow the solution space maximally with minimal overhead.

While we have shamelessly positioned SMT solving as reasonable efficient while exceedingly expressive relative to peers, the main message of the illustration is that SMT technologies have been developed, especially since the 2000s, with an emphasis on software applications and borrowing and stealing techniques that have otherwise been perfected by peer technologies. This is exemplified by the fact that Z3's main solver is based on a SAT solver CDCL architecture; it uses dual simplex and related global in-processing techniques from SAT solving to take advantage of global inferences; and finally, our use case illustrates incorporating global propagators. A well-recognized competing foundation for integration of solvers is to leverage a MIP solver instead of a CDCL core. This foundation benefits from global inferences and having strong MIP. Lookahead solvers, developed in the SAT community in the 90s [19] and revitalized for cube and conquer solving in the past decade [20], share some of the same traits as global MIP inferences and are promising methods for partitioning harder problems for a setting with distributed solving.

The solution to virtual plant configurations we are going to describe relies on SMT techniques. In particular, we are going to leverage mostly propositional SAT solving, coupled with a core SMT theory, the theory of uninterpreted functions, and augmented with a CP-inspired plugin for propagating global constraints.

2. Virtual Plant Configurations

In the following we describe virtual plant configurations in sufficient detail to appreciate problem characteristics and the nuances involved. The plant configurations are *virtual*; several points in a design space are explored for planning final physical configurations. We do omit several details that don't introduce crucial different concepts. For example, our full model contains a notion of *sub-line*, which are line segments. It contains constraints that limit how processes can be assigned to common sub-lines. Otherwise, our presentation is purposefully somewhat low level to convey an idea of the number and nature of concepts required for domain engineering. We use the following main domains to describe production lines:

Station, *Line*, *Monument*, *Process*, *Task*, *Zone*, *Operator*

Production lines are specific instantiations of these domains. The set $Task$ is instantiated with $O(10K)$ different tasks and *Station* comprises of hundreds of stations.

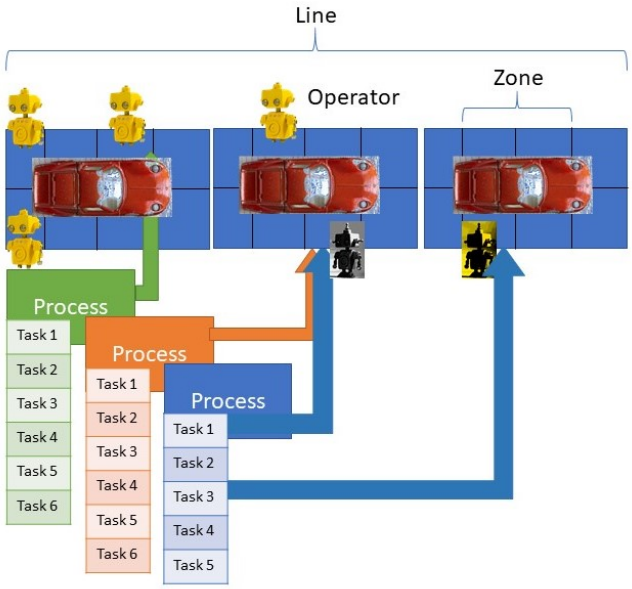

Figure 2. Elements of a Production Line

The virtual plant optimization problem is, in a nutshell, to assign each *Task* to a station and operator on the selected station. Thus, we are synthesizing two functions:

$$station : Task \rightarrow Station$$
$$operator : Task \rightarrow Operator$$

The assignment is subject to timing, capacity, and precedence constraints, and optimization objectives to minimize operator use, minimize station utilization, minimize tool utilization, and minimize height incompatibilities between tasks assigned to the same station.

Let us first describe the relevant domains and then give an idea of the hard constraints and optimization objectives.

2.1. Domains

2.1.1. Stations and Monuments Each station supports a subset of viable *operators* and has an associated timeout. Tasks assigned to the same operator on a station must be completed within the station timeout. The optional *monument* attribute is used to impose ordering between processes and stations. The reader can think of a monument as a coloring, and ordering constraints can be imposed

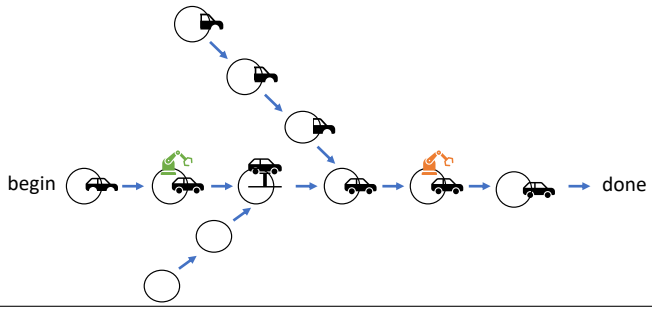

Figure 3. A fishbone assembly line. Each circle represents a station. Partially completed artifacts move between stations. The main line (in the center) carries partial cars where parts are attached to, while sub-lines flowing to the main line assemble smaller parts into smaller ones so that they reach the main line as a single part. For example, doors can be assembled in a sub-line and attached to the car in the main line. Each station has a different set of tools and machines.

on a set of stations with the same color. Stations are indexed by unique keys.

$$
\begin{array}{lllll}
\mathit{Station} & = & \langle & & \\
 & & \mathit{key} : & \mathit{Id} & \\
 & & \mathit{monument} : & [\mathit{Monument}] & \mathit{Optional\ monument\ tag} \\
 & & \mathit{next} : & [\mathit{Station}] & \mathit{Next\ station\ in\ line} \\
 & & \mathit{timeout} : & \mathit{Numeral} & \mathit{Station\ time\ bound} \\
 & & \mathit{line} : & \mathit{Line} & \mathit{Sub-line\ where\ station\ resides} \\
 & & \mathit{operators} : & \mathit{Operator}\textbf{-set} & \mathit{Viable\ operators\ on\ a\ station} \\
 & \rangle & & &
\end{array}
$$

Stations are organized in a tree structure, also referred to as a *fish-bone* structure. Figure 3 illustrates an abstract production line. The *next* attribute points to the successor station closer to the root of the tree. It is null if the station is last on the line.
For the purpose of this paper, monuments and lines are identified by a unique key.

$$
\begin{array}{lll}
\mathit{Monument} & = & \langle \mathit{key} : \mathit{Id} \rangle \\
\mathit{Line} & = & \langle \mathit{key} : \mathit{Id} \rangle
\end{array}
$$

With each monument, m, the set of stations tagged by m is given by:

$$\mathit{stations}(m) \ := \ \{\, s \in \mathit{Station} \mid s.\mathit{monument} = m \wedge m \neq \mathit{null} \,\}$$

2.1.2. Processes, Tasks, Zones, and Operators A process encapsulates a set of related tasks. Processes may be constrained in three ways: The *before* and *after* attributes are used to constrain processes to be assigned to stations before/after stations labeled by the given monuments. The *predecessor* attribute

imposes an ordering between processes and the *parallel* attribute identifies sibling processes that must be assigned to the same stations (e.g., one cannot fill coolant without also filling brake fluids). Processes may furthermore be labeled as under/over body exclusive when they can't be assigned to a station that contains both under and over-body work. Thus, we have:

$$
\begin{array}{lll}
Process = \langle & & \\
\quad key: & Id & \\
\quad ubx: & Bool & \\
\quad before: & [Monument] & Monument\ to\ precede \\
\quad after: & [Monument] & Monument\ to\ succeed \\
\quad predecessor: & [Process] & Process\ to\ succeed \\
\quad parallel: & Process\text{-}\mathbf{set} & Processes\ to\ co-assign \\
\rangle &
\end{array}
$$

Tasks are associated with a host process and characterized by a completion time, the height where the task is completed, a set of viable operators that are capable of servicing the task, and a flag *ub*, indicating whether the task is completed below the car.

$$
\begin{array}{lll}
Task = \langle & & \\
\quad key: & Id & \\
\quad process: & Process & Process\ where\ task\ belongs \\
\quad time: & Numeral & task\ execution\ time \\
\quad height: & Numeral & work\ height \\
\quad zone: & Zone & area\ where\ task\ takes\ place \\
\rangle &
\end{array}
$$

$$
\begin{array}{lll}
Zone = \langle & & \\
\quad key: & Id & \\
\quad operators: & Operator\text{-}\mathbf{set} & viable\ operators\ for\ zone \\
\quad ub: & Bool & is\ the\ zone\ upper\ or\ lower\ body \\
\rangle &
\end{array}
$$

The set of tasks associated with a process p is therefore given by:

$$tasks(p) := \{\ t \in Task \mid t.process = p\ \}$$

Finally, there is a finite small set of possible operators per station.

$$Operator = \{\ Op_1, \ \ldots Op_{10}\ \}$$

2.2. A Formalization of Domain Constraints

In the following we will describe a representative set of domain constraints. They capture *hard* constraints that must be met by physical or policy requirements, such as, one cannot attach a steering wheel before the dashboard is in place. The

formalization comes close to the working model, but leaves out a few details to preserve space. Precedence constraints capture ordering requirements between processes and between stations and processes. Operator constraints capture how tasks can be assigned to operators on assigned stations. Finally, cycle-time constraints bound the number and duration of tasks assignable to a station. To formulate the precedence constraints we will need a predicate that captures the partial order on stations.

2.2.1. Station Precedence Encoding The relation

$$\preceq\colon \mathit{Station} \times \mathit{Station} \to \mathit{Bool}$$

defines a partial (tree) order on stations. The ordering on stations can be encoded by introducing two sequence numbers: The first sequence number *leftOrd* is obtained by assigning sequence numbers following a depth-first, left-to-right order tree traversal, the other, *rightOrd* from a depth-first, right-to-left traversal. In other words, the traversal starts with the last station in the line, walks back and either branches to the left-most sub-line whenever two lines join or branches to the right-most sub-line. Then $\preceq$ is defined as:

$$s_2 \preceq s_1 := \mathit{leftOrd}(s_1) \leq \mathit{leftOrd}(s_2) \wedge \mathit{rightOrd}(s_1) \leq \mathit{rightOrd}(s_2)$$

2.2.2. Process precedence constraints For every task t we have the *monument ordering* constraints that confine tasks between monuments:

$$\min \mathit{stations}(t.\mathit{process}.\mathit{after}) \preceq \mathit{station}(t) \preceq \max \mathit{stations}(t.\mathit{process}.\mathit{before})$$

In words: a monument m may be associated with a set of stations S. A process $p := t.\mathit{process}$ that has m as the `before` monument, should be assigned to a station that is before, or including, the last station in S. A process p that has m as the `after` monument should take place after, or including, the first station in S. We conveniently assume $\min \emptyset = -\infty$ and $\max \emptyset = +\infty$ to deal with the cases where the *before* or *after* monument attributes are null.

The astute reader will note that it is also possible to have monument attributes, such as $t.\mathit{process}.\mathit{before}$, that are untethered. That is, there is no station tagged with the monument. For these monuments, the rule is that all processes associated with the same untethered monument reside on the same line, so we impose equations of the form

$$\mathit{station}(t_1).\mathit{line} = \mathit{station}(t_2).\mathit{line}$$

for cases such as $t_1.\mathit{process}.\mathit{before} = t_2.\mathit{process}.\mathit{before} \neq \mathit{null}$ with the requirement that $\mathit{stations}(t_1.\mathit{process}.\mathit{before}) = \emptyset$.

Similarly, a precedence relation is imposed by *predecessor* processes:

$$t_1.\mathit{process} = t_2.\mathit{process}.\mathit{predecessor} \Rightarrow \mathit{station}(t_1) \preceq \mathit{station}(t_2)$$

Parallel tasks have to be assigned the same stations:

$$t_1.process \in t_2.process.parallel \Rightarrow station(t_1) = station(t_2)$$

Tasks belonging to the same process are assigned to the same or at most two neighboring stations:

$$\begin{aligned} t_1.process = t_2.process \Rightarrow \quad & station(t_1) \in \{ station(t_2),\ station(t_2).next \} \\ \vee \quad & station(t_2) \in \{ station(t_1),\ station(t_1).next \} \end{aligned}$$

2.2.3. Operator constraints Tasks may only be assigned to stations that supports one of the assignable operators:

$$operator(t) \in t.zone.operators \cap station(t).operators$$

Tasks assigned the same zone on a station must use the same operator.

$$\begin{aligned} &(t_1.zone = t_2.zone \wedge station(t_1) = station(t_2)) \\ &\quad \Rightarrow operator(t_1) = operator(t_2) \end{aligned}$$

Tasks that are marked as under-body exclusive cannot be assigned to a station with tasks having conflicting zones:

$$\begin{aligned} &(t_1.process.ubx \wedge station(t_1) = station(t_2)) \\ &\quad \Rightarrow t_1.zone.ub \Leftrightarrow t_2.zone.ub \end{aligned}$$

At most 6 operators (preferably at least 2) can be assigned to a station:

$$2 \leq |\{ operator(t) \mid t \in Tasks,\ station(t) = s \}| \leq 6$$

The difference between the max and min height used at a station s is bounded (by 200mm):

$$\begin{aligned} &StationHeights(s) := \{ t.height \mid t \in Tasks,\ station(t) = s \} \\ &\max StationHeights(s) - \min StationHeights(s) \leq 200 \end{aligned}$$

2.2.4. Cycle-time constraints The time taken by tasks assigned in each operator zone on a station cannot exceed the station completion-time. To formulate this constraint, define the *opTime* of operator *op* on stations s as:

$$opTime(s,\ op) := \sum\{ t.time \mid t \in Task,\ station(t) = s,\ operator(t) = op \}$$

Then the cycle time constraints are, for every station s and $op \in s.operators$:

$$opTime(s,\ op) \leq s.timeout$$

2.3. Objectives

There is no single objective that governs as a metric for the quality of a production line. Instead there is a collection of objectives that are desirable. They are derived from reducing the cost and maximizing throughput of a production line. Costs are determined by the number of operators and the number of physical assets, stations, and tools; the main cost reduction objectives are therefore:

- Minimize overall number of operators used in a production line.
- Minimize overall number of utilized stations, that is, stations with a non-zero number of operators.
- Minimize overall number of different tools used for the production line.

Other auxiliary objectives are indirectly related to cost. For instance, avoiding lifting and lowering tools and cars between stations, contributes to a smoother operation with reduced risks for accidents.

- Minimize operator congestion on stations. A station is congested if it uses more than four operators.
- Minimize process fragmentation, that is minimize the number of processes that are split.
- Minimize the height differences of tasks within each station and between adjacent stations.
- Minimize operators that are used, but under-utilized on a station, i.e., the operator's assigned tasks can be completed in a small fraction of the station's overall timeout.

2.4. Solvable Formalizations

The formalization we just presented fully describes a set of admissible configurations. It is, however impractical to work with and a much more compact encoding is possible by taking advantage of characteristics of the model and by using specialized code to enforce constraints instead of creating large formulas.

2.4.1. Processes instead of Tasks An intrinsic property of the model is that tasks are naturally grouped by processes. The grouping is reflected in the admissible assignments: tasks belonging to the same process can only be assigned to at most two adjacent stations. The number of tasks is furthermore an order of magnitude larger than the number of processes. Thus, by formulating constraints by referring to processes instead of tasks saves roughly an order of magnitude constraints. So instead of solving for assigning tasks to stations we solve for assigning processes to stations, and independently determine whether processes are split. In the modified formulation we are therefore synthesizing the functions:

$$\begin{aligned} \mathit{station} &: \quad \mathit{Process} \to \mathit{Station} \\ \mathit{operator} &: \quad \mathit{Process} \times \mathit{Zone} \to \mathit{Operator} \end{aligned}$$

Operator assignment takes a work zone as argument to account for that different tasks within a process are allowed to be assigned different zones.

For processes where all tasks are assigned the same station, all properties of tasks are preserved. But for processes whose tasks are split between stations, the reformulation to processes reduces the solution space from the solver. For splittable processes we partition process tasks into two partitions $p.preTasks$ and $p.postTasks$, such that $p.preTasks \cup p.postTasks = tasks(p)$ and $p.preTasks \cap p.postTasks = \emptyset$.

Committing early on for whether processes can be split is a potential source of fragmentation: a solution may not be able to fully utilize station resources because processes are split while they can still utilize some station time. Characteristics of the production plant models come to the rescue, though. The vast majority of processes are relatively short running compared to station timeouts and any internal fragmentation resulting from restricting how they may be split is a smaller fraction of station timeouts.

To describe the process-based encoding we introduce a predicate:

$$\textit{isSplit}: \quad \textit{Process} \rightarrow \textit{Bool} \qquad\qquad \textit{Is process split between stations}$$

and require that split processes reside on the same line:

$$\textit{isSplit}(p) \Rightarrow \textit{station}(p).\textit{line} = \textit{station}(p).\textit{next}.\textit{line}$$

Constraints that are originally formulated using $\textit{station}(t)$, where t is a task, are now reformulated using processes, using $\textit{station}(p)$ for process p containing task t. Converting the encoding to use processes is relatively straight-forward, thus we omit it.

2.4.2. Uninterpreted functions to the rescue One approach to encode height constraints is to introduce two functions:

$$\begin{array}{lll} \textit{minHeight}: & \textit{Station} \rightarrow \textit{Nat} & \textit{Minimal height of tasks on a station} \\ \textit{maxHeight}: & \textit{Station} \rightarrow \textit{Nat} & \textit{Maximal height of tasks on a station} \end{array}$$

and then impose

$$\begin{array}{ll} \textit{minHeight}(\textit{station}(t)) \leq t.\textit{height} \leq \textit{maxHeight}(\textit{station}(t)) & \forall t \in \textit{Task} \\ \textit{maxHeight}(s) - \textit{minHeight}(s) \leq 200 & \forall s \in \textit{Station} \end{array}$$

If we did not have functions to our disposal, and instead used two variables $s.\textit{minHeight}, s.\textit{maxHeight}$ per station s, we would have to formulate the bounds on $s.\textit{minHeight}$ and $s.\textit{maxHeight}$ using $|Task| \times |Station|$ constraints of the form:

$$\textit{station}(t) = s \Rightarrow s.\textit{minHeight} \leq t.\textit{height} \leq s.\textit{maxHeight},$$

for each task t and station s.

By using uninterpreted functions we only assert $|Task|$ constraints to enforce each min-height bound. With hundreds of stations, this saves two orders of magnitudes in the encoding. Furthermore, as we are also only indirectly encoding

station(t) by instead using an assignment of stations on processes, we save another order of magnitude in terms of number of constraints. The process-based encoding, thus takes the form:

$$\begin{array}{l}
\neg \mathit{isSplit}(p) \;\Rightarrow \mathit{minHeight}(\mathit{station}(p)) \;\leq \min\{\; t.\mathit{height} \mid t \in \mathit{tasks}(p) \;\} \\
\mathit{isSplit}(p) \;\Rightarrow \mathit{minHeight}(\mathit{station}(p)) \;\leq \min\{\; t.\mathit{height} \mid t \in p.\mathit{preTasks} \;\} \\
\mathit{isSplit}(p) \;\Rightarrow \mathit{minHeight}(\mathit{station}(p).\mathit{next}) \;\leq \min\{\; t.\mathit{height} \mid t \in p.\mathit{postTasks} \;\}
\end{array}$$

and symmetrically for *maxHeight*.

2.4.3. Avoiding pairwise constraints Modeling with uninterpreted functions comes with some useful tricks of the trade. For example, if we wish to enforce that a function f is injective, it can be encoded by requiring for every pair of argument combination x, y:

$$f(x) \;=\; f(y) \;\Rightarrow x \;=\; y$$

But a much more succinct encoding uses an auxiliary partial inverse function g with constraints

$$g(f(x)) \;=\; x$$

for every x. This can have a dramatic effect if the domain of f is large; say the number of tasks is $O(10K)$, then the pairwise encoding requires $O(100M)$ constraints. A phenomenon related to injectivity surfaces when encoding zone assignments and under-body mutual exclusion. Recall the requirements

$$\begin{array}{l}
(t_1.\mathit{zone} \;=\; t_2.\mathit{zone}) \wedge (\mathit{station}(t_1) \;=\; \mathit{station}(t_2)) \;\Rightarrow \mathit{operator}(t_1) \;=\; \mathit{operator}(t_2) \\
t_1.\mathit{process}.\mathit{ubx} \wedge (\mathit{station}(t_1) \;=\; \mathit{station}(t_2)) \;\Rightarrow (t_1.\mathit{zone}.\mathit{ub} \;\Leftrightarrow t_2.\mathit{zone}.\mathit{ub})
\end{array}$$

They consider all pairs of tasks. The first requirement can be captured more succinctly by introducing a predicate that tracks which work zones are used on a station and a function $wz2op$ that assigns operators to work zones on stations. The second can be handled using a similar idea that uses a predicate that tracks whether a station is assigned an under-body exclusive task.

Thus, for each process p:

$$\begin{array}{lll}
\neg \mathit{isSplit}(p) & \Rightarrow \mathit{wzUsed}(\mathit{station}(p),\; z) & \forall z \in \{\; t.\mathit{zone} \mid t \in \mathit{tasks}(p) \;\} \\
\mathit{isSplit}(p) & \Rightarrow \mathit{wzUsed}(\mathit{station}(p),\; z) & \forall z \in \{\; t.\mathit{zone} \mid t \in p.\mathit{preTasks} \;\} \\
\mathit{isSplit}(p) & \Rightarrow \mathit{wzUsed}(\mathit{station}(p).\mathit{next},\; z) & \forall z \in \{\; t.\mathit{zone} \mid t \in p.\mathit{postTasks} \;\}
\end{array}$$

If the process has $p.ubx$ set to true, we add also:

$$\begin{array}{lll}
\neg \mathit{isSplit}(p) & \Rightarrow \mathit{wzUbx}(\mathit{station}(p),\; z) & \forall z \in \{\; t.\mathit{zone} \mid t \in \mathit{tasks}(p) \;\} \\
\mathit{isSplit}(p) & \Rightarrow \mathit{wzUbx}(\mathit{station}(p),\; z) & \forall z \in \{\; t.\mathit{zone} \mid t \in p.\mathit{preTasks} \;\} \\
\mathit{isSplit}(p) & \Rightarrow \mathit{wzUbx}(\mathit{station}(p).\mathit{next},\; z) & \forall z \in \{\; t.\mathit{zone} \mid t \in p.\mathit{postTasks} \;\}
\end{array}$$

Note that since practically all tasks associated with each process share the same zone, there are in the common case only three constraints per process for *wzUsed*, and for *wzUbx*, respectively.

For station s and each work zone z

$$\begin{aligned} wzUsed(s,\ z)\ \Rightarrow\ & wz2op(s,\ z)\ \in z.operators \cap s.operators \\ \wedge\ & wzUbx(s,\ z)\ \Rightarrow (ubUsed(s)\ \Leftrightarrow z.ub) \end{aligned}$$

Note how the predicate $ubUsed(s)$ gets constrained to be true if $z.ub$ is true and $wzUbx(s, z)$ is implied based on some task occupying the workzone z on station s.

2.4.4. Cycle time constraints as code Finally, we omit encoding cycle time constraints entirely in our formulation. A major issue with fully expanding cycle time constraints is that it requires in the worst case to include the possibility that each task is assigned to every possible station and operator zone. Thus, it requires $|Station| \times |Operators|$ constraints each adding up $|Task|$ terms. Section 4.4 describes our encoding of cycle time constraints as a custom propagator using an API of Z3 that allows encoding constraints as code.

3. Experiences with Domain Engineering

Section 2 described a formalization of virtual plant configurations. Let us describe how the formalization was used to debug virtual plan configurations. Instances of virtual plant configurations are stored in SQL tables. Enforcing the domain constraints is well outside the scope of domain-agnostic database consistency guarantees, but we can take a software-inspired view and treat configurations as code and check invariants as if we are checking assertions of software.

3.1. Model visualization

The value of model visualization is very well recognized in the CP and model-based development communities [33]. The MSAGL tool [26] was initially developed to support model-based software development using abstract state machines [3], but has since been used broadly, such as in Visual Studio [30]. In our case, graph visualization proved to be an effective way to communicate how a virtual plant model in a database is interpreted in a formal model.

3.2. Checking Global Model Invariants

Initial experiments with visualization suggested that the virtual plant representation in the database did not contain sufficient information to reproduce a physically connected production plant. Omitted data-entries or data-entry errors would render product sub-lines disconnected. Similarly, precedence relations between processes could end up being cyclic as a result of data-entry errors. The situation is analogous to software development: a type checker can catch a large class of unsafety bugs cheaply.

A common type of bugs we encountered was in the processes' precedence relations. We found several cases where a process preceding another process

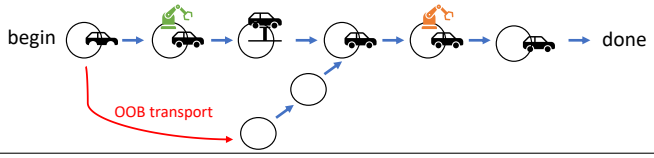

Figure 4. Out-of-band transportation of parts between the main line and sub-lines. We cannot have a precendence relation between the producer and consumer of these parts as these processes run in parallel lines.

was supposed to run in a parallel sub-line. This is not possible as stations in parallel sub-lines have no precedence relation between them (ordering of stations is partial). This was caused by a confusion when the data was entered. These processes effectively run one before the other in a deployed production line if we consider the sub-lines side-by-side. However, there was no process precedence; it was just an artifact of the current solution.

Another kind of precedence bugs we found was related with processes that are not explicitly modeled. For example, some parts are removed from the chassis of the car in one station. Then they are transported on the side to a subsequent station where they get re-attached (Figure 4). The transportation of these parts is not modeled because we know it can be done in a timely fashion and does not happen in the main conveyor belt, which is what we model. However, initially the processes that receive the removed parts had precedence on the processes that remove these parts, even when the receiving processes were in stations what were not successors of the removing stations. The fix was to remove the precedence relation and consider it on paper only (i.e., the process engineers have to ensure the out-of-band transportation can be done in a timely fashion).

3.3. Root-cause analysis using Unsatisfiable Cores

Global invariants only ensure that solutions to satisfiable constraint encodings correspond to feasible plant configurations. They don't ensure that constraints are feasible. Infeasible constraints are as inevitable as software bugs: they originate from manual data-entry errors that are difficult to avoid because consistency is a global property involving thousands of entries. Bug localization using unsatisfiable cores and program repair using correction sets is already well recognized [22, 31, 36]. In Figure 5 we show an example of an unsatisfiable core that was encountered in one of our runs. It involves chaining several equalities and arriving at the equality $1 = 3$.

To make the inconsistency palatable in terms of concepts used in the data-model, we tracked each assertion by originator information and used this to produce an error report that could be digested at the level of the model, as opposed to the raw encoding. Figure 6 illustrates the same unsatisfiable core, but rendered from the perspective of the database.

```
line(103) == 1
line(119) == 3
line(station(WHEEL ASSEMBLY INSTALL FR RH)) == line(119)
line(station(LIFTGATE LATCH TO LIFTGATE INSTALL)) == line(103)
line(station(WHEEL CAP INSTALL - FR LH)) ==
     line(station(WHEEL ASSEMBLY INSTALL FR RH))
line(station(WHEEL CAP INSTALL - FR LH)) ==
     line(station(LIFTGATE SEAL - RIP CORD LH))
line(station(LIFTGATE SEAL - RIP CORD LH)) ==
     line(station(LIFTGATE LATCH TO LIFTGATE INSTALL))
```

Figure 5. Unsatisfiable core from Z3

```
stations [TR1-240-R-N] are on line [TR]
stations [FN1-150-R-N] are on line [FN]
processes [WHEEL ASSEMBLY INSTALL FR RH] must be assigned to the
  same line as their monuments at [FN1-150-R-N]
processes [WHEEL CAP INSTALL - FR LH] sharing untethered monument
  [WHEEL ASSEMBLY INSTALL FR RH, LIFTGATE SEAL - RIP CORD LH] must
  be assigned to the same line
processes [LIFTGATE LATCH TO LIFTGATE INSTALL] must be assigned to the
  same line as their monuments at [TR1-240-R-N]
processes [LIFTGATE SEAL - RIP CORD LH] sharing untethered monument
  [LIFTGATE LATCH TO LIFTGATE INSTALL] must be assigned to the same line
```

Figure 6. Explanation From Unsat Core

Z3 uses MiniSAT's approach [16] by using tracking literals to extract unsatisfiable cores. Cores are optionally minimized using a greedy core minimization algorithm that forms the basis of SAT-based MUS extraction tools [4].

4. Experiences With Solver Engineering

We will be describing the elements used in our current solver. It finds feasible solutions to production lines within a couple of minutes and then yields optimized solutions in a steady stream as the solver explores Pareto fronts. The journey to our current approach took several iterations. During initial iterations, finding just one feasible solution was elusive.

We tried three conceptually different approaches, prior to the eventual solution we describe next. These approaches were differentiated by how they attempted to address the special complexity of cycle-time constraints.

- A first approach created an encoding of all domain constraints, except cycle-time constraints. Then the solver would assign a small batch of processes by adding cycle-time constraints at a time. The approach scaled to less than a dozen processes per batch and it took around 20 hours to solve for 10% of all processes.
- In a second experiment we added cycle-time constraints for processes one by one, and greedily assigned them to stations. The experiment relied on auxiliary static analysis to narrow the range of possible station assignments for every process and we would prioritize processes with the narrowest range of feasible stations. With this approach we could assign 80% of processes using 10 hours CPU processor time.
- A third experiment aimed to build a CP engine on top of Z3 by augmenting the greedy approach with backtracking so that it could assign all processes. The idea was that the external CP engine would make branching decisions on how to assign processes and also manage backtracking. While engineering this approach was too complex to fully realize, it served as a guide for the approach we arrived at with *constraints as code*. Here, branching decisions remain inside of Z3, but conflict detection and theory propagation is programmed by a CP module for cycle-time constraints.

4.1. SMT Theories and Solvers

Z3 supports a rich collection of formalisms that go well beyond the features used in this work. It supports theories of bit-vectors, uninterpreted functions, arrays, algebraic datatypes, floating points, strings, regular expressions, sequences, bounded recursive function unfolding and partially ordered relations. To support the many formalisms and different classes of formulas Z3 contains a plethora of powerful engines. A CDCL(T) core glues together most supported theories in a combined reasoning engine. The core also integrates with quantifier instantiation engines. Other reasoning cores can be invoked in stand-alone ways, including a

core for non-linear real Tarskian arithmetic, decidable quantified theories, and a Horn clause solver [12]. The work described in this paper draws on only a few of the available formalisms and engines: bit-vectors and uninterpreted functions. Central to the art of solver engineering is choosing the best theories and encoding for a particular problem.

4.2. Uninterpreted functions

We already mentioned that we use the theory of *uninterpreted functions*, also known as EUF. It is basic to first-order logic and treated as a base theory for SMT solvers. EUF admits efficient saturation using congruence closure algorithms [14]. Uninterpreted functions are well recognized in SMT applications as useful for abstracting data [13] and in model checking of hardware designs [2]. Congruence closure consumes a set of equalities over terms with uninterpreted functions and infers all implied equalities over the terms used in the equalities. Consider for example, the two equalities

$$x = f(g(f(x))),\ x = g(f(x))$$

We can use the second equality to simplify the first one: by replacing the sub-term $g(f(x))$ in $f(g(f(x)))$ by x, the first equality reduces to $x = f(x)$. This new equality can be used to simplify the second equality by replacing the sub-term $f(x)$ in $g(f(x))$ by x. The resulting equality is $x = g(x)$. Congruence closure algorithms perform such inferences efficiently, without literally substituting terms in equations.

Using EUF instead of encoding directly into SAT is not necessarily without a cost. By default, SMT solvers allow only inferences over EUF that do not introduce new terms. This prevents the solvers from producing short resolution proofs in some cases, but has the benefit of avoiding bloating the search space with needless terms. Efficient solvers seek a middle-ground by introducing transitive chaining of equalities and Ackerman reductions on demand [9, 15]. For the use case described in this paper, even these on-demand reductions turn out to be harmful and slow down search. They are disabled for this application. Furthermore, we found it useful to delay restarts to give the solver time to perform model-repair in contrast to producing resolvents. Precisely how to tune SAT solvers for satisfiable instances is a topic [7, 27, 29] where new insights are currently developed.

4.3. Bit-vectors

The first few encoding attempts used the theory of arithmetic and integers to represent all domains. While not exclusively responsible for inferior performance, we noticed an order of magnitude speedup on the same formulations when switching to bit-vectors. Finite domains can be encoded directly using bounded integers. The usual ordering $\leq$ on integers can then be used whenever requiring precedence relations or comparing heights. Except for cycle-time constraints that we deal

with separately, there is however very little or practically no arithmetic involved with the constraints. By using bit-vectors instead of integer data-types we can force Z3 to use bit-vector reasoning for finite domains. The theory of bit-vectors is used to capture machine arithmetic, with noteworthy applications for analysis of binary code or compiler intermediary languages, thus two-complements arithmetical operations over 32-bit or 64-bit arithmetic found in machine code. Comparison, $\leq$ is defined for both signed and unsigned interpretations of bit-vectors. These operations are used extensively for modeling operator precedence and height constraints. The bit-vector representation and reasoning was order of magnitudes more efficient than using encoding relying on arithmetic. It conforms to common experiences where using arithmetic for finite domain combinatorial problems is rarely an advantage. Mainstream SMT solvers solve bit-vectors by a reduction to propositional SAT. It works well for this domain, in contrast to constraints involving multiplication of large bit-vectors. Handling larger bit-widths is a long standing open challenge for SMT solvers.

4.4. Constraints as Code

Early experiments suggested that adding Pseudo-Boolean inequalities corresponding to cycle-time constraints would be a show-stopper. It is an instance where existing built-in features do not allow for a succinct encoding. These constraints highlighted a need for exposing a flexible approach for extending Z3 with ad-hoc, external, theory solvers. Z3 exposed a way for encoding external solvers more than a decade ago [8]. External theories were subsequently removed from Z3 because not all capabilities of internal theory solvers could be well supported for external solvers. Moreover, with Z3 being open source, the path was prepared for external contributions, such as Z3Str3 [5]. But we found that the cycle-time constraints are not easily amenable to a new theory; the conditions for when they propagate consequences or identify conflicts depend on properties that are highly specific to this particular model. It is thus much easier to represent propagation and conflict detection in code than in constraints.

We will illustrate the user propagator by a simple example borrowed from [10]. It illustrates a Pseudo-Boolean constraint that requires a quadratic size encoding. In contrast, the user propagator does not suffer from this encoding overhead. The example constraint is:

$$\begin{aligned} &3\,|\{\,(i,j)\mid i<j\wedge x_i+x_j=42\wedge(x_i>30\vee x_j>30)\,\}| \\ &+\,|\{\,(i,j)\mid i<j\wedge x_i+x_j=42\wedge x_i\leq 30\wedge x_j\leq 30\,\}| \qquad \leq 100 \end{aligned}$$

For illustration, we instantiate the example with 8 bit-vectors each with 10 bits over Python:

```
from z3 import *

xs = BitVecs(["x%d" % i for i in range(8)], 10)
```

Then a user-propagator can be initialized by sub-classing to the `UserPropagateBase` class that implements the main interface to Z3's user propagation functionality.

```
class UserPropagate(UserPropagateBase):
    def __init__(self, s):
        super(self.__class__, self).__init__(s)
        self.add_fixed(self.myfix)
        self.add_final(self.myfinal)
        self.xvalues = {}
        self.id2x = { self.add(x) : x for x in xs }
        self.x2id = { self.id2x[id] : id for id in self.id2x }
        self.trail = []
        self.lim = []
        self.sum = 0
```

The map `xvalues` tracks the values of assigned variables and `id2x` and `x2id` maps tracks the identifiers that Z3 uses for variables with the original variables. The `sum` maintains the running sum of according to our unusual constraint.

The class must implement methods for pushing and popping backtrackable scopes. We use a `trail` to record closures that are invoked to restore the previous state and `lim` to maintain the the size of the trail for the current scope.

```
    # overrides a base class method
    def push(self):
        self.lim.append(len(self.trail))

    # overrides a base class method
    def pop(self, num_scopes):
        lim_sz = len(self.lim)-num_scopes
        trail_sz = self.lim[lim_sz]
        while len(self.trail) > trail_sz:
            fn = self.trail.pop()
            fn()
        self.lim = self.lim[0:lim_sz]
```

We can then define the main callback used when a variable tracked by identifier `id` is fixed to a value `e`. The identifier is returned by the solver when calling the function `self.add(x)` on term `x`. It uses this identifier to communicate the state of the term `x`. When terms range over bit-vectors and Booleans (but not integers or other types), the client can register a callback with `self.add_fixed` to pick up a state where the variable is given a full assignment. For our example, the value is going to be a bit-vector constant, from which we can extract an unsigned integer into `v`. The trail is augmented with a restore point to the old state and the summation is then updated and the Pseudo-Boolean inequalities are then enforced.

```
    def myfix(self, id, e):
        x = self.id2x[id]
        v = e.as_long()
```

```
        old_sum = self.sum
        self.trail.append(lambda : self.undo(old_sum, x))
        for w in self.xvalues.values():
            if v + w == 42:
                if v > 30 or w > 30:
                    self.sum += 3
                else:
                    self.sum += 1
        self.xvalues[x] = v
        if self.sum > 100:
            self.conflict([self.x2id[x] for x in self.xvalues])
        elif self.sum < 10 and len(self.xvalues) > len(xs)/2:
            self.conflict([self.x2id[x] for x in self.xvalues])
```

It remains to define the last auxiliary methods for backtracking and testing.

```
    def undo(self, s, x):
        self.sum = s
        del self.xvalues[x]


    def myfinal(self):
        print(self.xvalues)


s = SimpleSolver()
for x in xs:
    s.add(x % 2 == 1)
p = UserPropagate(s)
s.check()
print(s.model())
```

4.5. Solving for multiple objectives

Z3 supports optimization modulo theories out of the box [11], including weighted MaxSAT and optimization of linear objectives. It can also be instructed to enumerate Pareto fronts or combine objectives through a lexicographic combination. In our case we are not, at present, using these features for optimizing objectives. Instead, we built a custom Pareto optimization mechanism on top of the user propagator. It is inspired by the branch-and-bound method for MaxSAT from [28]. The idea is that each objective function is registered with an independent constraint handler. Each handler maintains a current cost. The current cost is incremented when a variable gets fixed in a way that adds to the running cost. For example, when a task is assigned a station, the tool used by the task is added to the pool of tools used, unless the tool is already used at the station. When then number of used tools exceeds the current running best bound for tools, the

handler registers a conflict. Handlers may also cause unit propagation when the current bound is reached. This approach has the benefit from producing partial results as soon as they are available. Several improvements are possible over this scheme, such as neighborhood search around current solutions. We leave this for future explorations, as the current approach is sufficient within the generous time budget for the plant configuration domain.

5. Experiences with MiniZinc

We also developed a plant model in MiniZinc. Following MiniZinc best practices, we used so-called global constraints that deal with functions/relations as first-class values, hence avoiding quantified constraints over individual elements. First, we used a channel constraint to connect a function that assigns a station to a process with its inverse (given that the inverse is used for various aggregations over station's processes). This channel dramatically improved the solving performance, compared to its equivalent formulation using universal quantification. Furthermore we used a bin packing constaint to ensure fit of processes into a station. Finally, we had to address the shortcoming of global constraints that they cannot be driven by decision variables and hence require an eager case distinction as a work-around. To reduce the ranges of decision variables participating the bin packing constraint, i.e., to avoid assuming that any process could be placed in *any* station, we developed an abstract-interpretation style approximation of the set of stations for a given process. Technically, this approximation is computed iteratively as the least-fixpoint of the propagation operator manually derived from the constraints.

We observed that the resulting performance with the Gecode solver backend is comparable with the Z3. We left it for future work to automate the construction of approximation operators.

6. Perspective

We described our experiences with using SMT and CP techniques for solving virtual plant configurations for production plants. The domain shares characteristics of job-shop scheduling and constrained knapsack problems. The scenario integrates a plethora of side constraints. Our perspective in tackling this domain is heavily influenced by methods adapted in the software-engineering, and particularly model-driven engineering and formal methods communities. Several synergies with configuration domains and advances in software engineering communities seem ripe to be explored: Automated software synthesis has gained considerable traction in the software engineering community [1]. SMT and SAT solvers are some of the popular options for handling software synthesis and *program sketching* problems. Super-compilation can be recast as a quantifier instantiation problem and template-based methods use a template space defined by abstract grammars to define a search space for synthesis problems. CVC4 [32] builds in grammar based synthesis as an extension of its quantifier instantiation

engine; efficient, custom, synthesis tools such as Prose [18], Rosette [37], and for program sketching [35], integrate specialized procedures.

Our SMT solution is based on Z3 with uninterpreted functions, bit-vectors and user-programmed constraint propagators. The virtual plant configuration solver is currently actively used for planning next generation production facilities. There are still many exciting avenues to pursue for super-charging virtual plant configurations, or network cloud configurations and policies for that matter: methodologies and tools developed for programming languages have substantial potential to transform configuration management; configurations can be improved using feedback measurements from deployments; and symbolic solving have a central role in checking integrity constraints, and synthesizing solutions while exploring a design space.

References

1. Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015. doi:10.3233/978-1-61499-495-4-1.
2. Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Reveal: A formal verification tool for verilog designs. In *LPAR*, 2008. doi:10.1007/978-3-540-89439-1_25.
3. Michael Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Towards a tool environment for model-based testing with AsmL. In *FATES*, 2003. doi:10.1007/978-3-540-24617-6_18.
4. Anton Belov and João Marques-Silva. Muser2: An efficient MUS extractor. *J. Satisf. Boolean Model. Comput.*, 8 (3/4): 123–128, 2012. doi:10.3233/sat190094.
5. Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: A string solver with theory-aware heuristics. In *FMCAD*, 2017. doi:10.23919/FMCAD.2017.8102241.
6. Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. Everest: Towards a verified, drop-in replacement of HTTPS. In *SNAPL*, volume 71 of *LIPIcs*, pages 1:1–1:12, 2017. doi:10.4230/LIPIcs.SNAPL.2017.1.
7. Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.
8. Nikolaj Bjørner. Engineering theories with Z3. In *APLAS*, 2011. doi:10.1007/978-3-642-25318-8_3.
9. Nikolaj Bjørner and Leonardo Mendonça de Moura. Tractability and modern satisfiability modulo theories solvers. In *Tractability: Practical Ap-*

proaches to Hard Problems, pages 350–377. Cambridge University Press, 2014. doi:10.1017/CBO9781139177801.014.

10. Nikolaj Bjørner and Lev Nachmanson. Navigating the universe of Z3 theory solvers. In *SBMF*, 2020. doi:10.1007/978-3-030-63882-5_2.
11. Nikolaj Bjørner and Anh-Dung Phan. νZ - maximal satisfaction with Z3. In *SCSS*, volume 30 of *EPiC Series in Computing*, pages 1–9. EasyChair, 2014. URL https://easychair.org/publications/paper/xbn.
12. Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M. Wintersteiger. Programming Z3. In *SETSS*, 2018. doi:10.1007/978-3-030-17601-3_4.
13. Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, 1994. doi:10.1007/3-540-58179-0_44.
14. Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27 (4): 758–771, 1980. doi:10.1145/322217.322228.
15. Bruno Dutertre. Yices 2.2. In *CAV*, 2014. doi:10.1007/978-3-319-08867-9_49.
16. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, 2003. doi:10.1007/978-3-540-24605-3_37.
17. Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55 (3): 40–44, 2012. doi:10.1145/2093548.2093564.
18. Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4 (1-2): 1–119, 2017. doi:10.1561/2500000010.
19. Marijn Heule and Hans van Maaren. Look-ahead based SAT solvers. In *Handbook of Satisfiability*, volume 185, pages 155–184. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-155.
20. Marijn J. H. Heule and Oliver Kullmann. The science of brute force. *Commun. ACM*, 60 (8): 70–79, 2017. doi:10.1145/3107239.
21. Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C. Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In *SIGCOMM*, 2019. doi:10.1145/3341302.3342094.
22. Manu Jose and Rupak Majumdar. Bug-assist: Assisting fault localization in ANSI-C programs. In *CAV*, 2011. doi:10.1007/978-3-642-22110-1_40.
23. Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *PLDI*, 2015. doi:10.1145/2737924.2737965.
24. Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded translation validation for LLVM. In *PLDI*, 2021. doi:10.1145/3453483.3454030.
25. Microsoft. Microsoft dynamics. https://dynamics.microsoft.com, 2021.
26. Lev Nachmanson. Microsoft automated graph layout tool. https://github.com/microsoft/automatic-graph-layout, 2021.
27. Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *SAT*, 2018. doi:10.1007/978-3-319-94144-8_7.
28. Robert Nieuwenhuis and Albert Oliveras. On SAT modulo theories and optimization problems. In *SAT*, 2006. doi:10.1007/11814948_18.
29. Chanseok Oh. Between SAT and UNSAT: the fundamental difference in CDCL SAT. In *SAT*, 2015. doi:10.1007/978-3-319-24318-4_23.

30. Sergey Pupyrev, Lev Nachmanson, Sergey Bereg, and Alexander E. Holroyd. Edge routing with ordered bundles. *Comput. Geom.*, 52: 18–33, 2016. doi:10.1016/j.comgeo.2015.10.005.
31. Raymond Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32 (1): 57–95, 1987. doi:10.1016/0004-3702(87)90062-2.
32. Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. CVC4SY for sygus-comp 2019. *CoRR*, abs/1907.10175, 2019. URL http://arxiv.org/abs/1907.10175.
33. Georg Sander and Adrian Vasiliu. The ILOG JViews graph layout module. In *GD*, 2002. doi:10.1007/3-540-45848-4_35.
34. João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48 (5): 506–521, 1999. doi:10.1109/12.769433.
35. Armando Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15 (5-6): 475–495, 2013. doi:10.1007/s10009-012-0249-7.
36. André Sülflow, Görschwin Fey, Roderick Bloem, and Rolf Drechsler. Using unsatisfiable cores to debug multiple design errors. In *VLSI*, 2008. doi:10.1145/1366110.1366131.
37. Emina Torlak and Rastislav Bodík. Growing solver-aided languages with rosette. In *SPLASH*, 2013. doi:10.1145/2509578.2509586.